Implementation and Optimization of LZW Compression Algorithm Based on Bridge Vibration Data

Fengyuan Zhang\textsuperscript{a}, Zhao Li\textsuperscript{b}, Mengli Chen, Lili Wen, Xiaolu Jia, Cheng Chen,\textsuperscript{a*}

\textsuperscript{a}Associate professor, Beijing University of Chemical And Technology, Beijing, 100029, China
\textsuperscript{b}Nongraduate, Beijing University of Chemical And Technology, Beijing, 100029, China

Abstract

LZW algorithm is adopted in this paper to compress the bridge vibration data. Based on the data structure of chained lists, an improved version of the algorithm, in which a new method called forward-moving on frequently-used entries is applied, is implemented and optimized. It can be drawn that using the above-mentioned method can effectively improve the searching speed and save compression time. In addition, optimizing the location of forward-moved frequently-used entries and selecting the appropriate statistical frequency have better optimization effects. Finally, implemented with C language, the algorithm is proved to have high compression efficiency with less time cost in the practical compression of bridge vibration data.

© 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.

Selection and/or peer-review under responsibility of [CEIS 2011]

Keywords: bridge vibration; LZW; chained lists; compression time; compression efficiency; statistic

1 Introduction

In recent years, bridge accidents occur frequently, which leads to higher requirements on the real-time monitoring on bridge vibration. The bridge vibration data from sensors is very large. And, in the practical application, Transmission Over Radio is always adopted to transmit the data to upper computer so that to monitor the real situation. While, the TOR transmission time is too long to achieve real-time monitoring...
purpose. Therefore, before transmitted, the huge data must be compressed [1]. Whereas the bridge vibration data is huge and have no regularity, this paper adopt LZW algorithm that is suitable to the data.

The original Lempel Ziv approach to data compression was first published in 1977, followed by an alternate approach in 1978. Terry Welch's refinements to the 1978 algorithm were published in 1984. The algorithm is surprisingly simple and is applied extensively. In a nutshell, LZW compression replaces strings of characters with single codes [2]. LZW can be implemented based on arrays, chained lists or tree. While being implemented by chained lists, and in the progress of building dictionary table, the algorithm will search for the codes with sequential traversal. Therefore, it may cost a little long time to find the codes, which will make the whole compression time get long [2]. To solve this problem, the paper use chained lists to implement LZW, and add a new variable to each node to keep count of the codes' amount of usage. When building a table, the high frequency codes are moved and inserted to the front end of the chained lists in order to save search time. Furthermore, through optimizing the front-insert location and adopting suitable statistic number, the optimization will have a better effect.

This paper's main content contains: Chapter2 describes the principles and features of LZW, Chapter3 presents the process of LZW's implementation and optimization, Chapter 4 draws the conclusion.

2 Principles and features of LZW

At the beginning, the dictionary contains only single characters and their code. Then start the compression, read new string, match it to the strings have been in the dictionary until it cannot match further. For the string matched successfully, output its corresponding code. Then combine this string with the character which causes the match failed together as a new string, and add it to the string table with the corresponding code. LZW string table is a table having prefix, that is to say each prefix of the string is also in the table. The dictionary is dynamically generated. In the process of extraction, the decoder can recover a same dictionary on the base of the codes and output the encoded data stream [3].

2.1 Principles of the LZW algorithm

The LZW algorithm’s coding steps are as follows [4]:

Step1: Initialize dictionary. Dictionary contains all single characters in the data stream.
Step2: Set the Prefix P null.
Step3: Read the next character in the data stream as the current character C.
Step4: Judge whether the string P + C is in the current dictionary.
   1) Yes, set P = P + C, that is extending P with C.
   2) No.
      ① output P’s corresponding code to the encoded data stream.
      ② Judge whether the dictionary achieve to the maximum capacity:
         If it doesn’t, add the string P + C to the dictionary, otherwise don’t do that.
      ③ Define P = C (P only contains C right now)
Step5: Judge whether there are characters in the data stream:
   1) Yes, return step3 to continue the encoding process.
   2) No, output P’s corresponding code to the encoded data stream.

End.

LZW decoding algorithm is the reverse process of encoding. The steps are as follows.
Step1: Initialize the dictionary, the dictionary contains all the single characters of data stream.
Step2: Read the first codeword CW in the encoded data stream.
Step3: Output CW’s corresponding string to the decoded data stream.
Step4: PW = CW, read the next codeword CW.
Step5: Judge whether there is a CW in dictionary:
1) Yes.
   ① Output the string $cW$ to the decoded data stream.
   ② $P = String.pW$.
   ③ $C =$ the first character of $String.cW$.
   ④ Add string $P + C$ to the dictionary.
2) No.
   ① $P = String.pW$.
   ② $C =$ the first character of $String.pW$.
   ③ Output string $P + C$ to decoded data stream and add it to the dictionary.

Step6: Judge whether there are code words in the encoded data stream.
1) Yes, return Step2 to continue decoding.
2) No, end decoding.

2.2 Features of LZW

LZW compression has good effect on the data which is unpredictable. And it has a high compression efficiency for consecutive bytes and strings in the data stream. It is a lossless compression so that the document is exactly the same as what it was before compression, and there are no any errors on every bit. In this way, it completely maintained the characteristics of the original document. It has a widely application during the field of data compression in image data processing and documents etc. The speed of compression and decompression is faster than any other algorithms. It can run in the Intel80386 computer [5].

3 The process of LZW’s implementation and optimization

3. 1 The basic implementation of LZW

The original bridge vibration data are denary decimal, whose length is long. What’s more, the data have little repeatability and no regularity. Therefore, the original vibration data need to be preprocessed at first. That is, the data should be transformed into binary digit before be compressed. Afterwards, on the base of unidirectional chained lists, the LZW algorithm compression and extraction are implemented with C programming language. The initial dictionary contains three parts. They are ’0’, ’1’ and ’.’. The compression is a process that build the dictionary dynamically while compress the raw data. When new entries need to be added to the dictionary, they should be inserted at the tail of the list. The process is shown in Fig.1. The compression and extraction dictionary are both fixed-length, and the length is 256. Searching for the codes in the dictionary adopt sequential traversal. Implemented algorithm could fullfill the requirement for bridge vibration data and its compression efficiency is relatively high. However, because of sequential traversal in the search, each time spending on it is long so that to make compression time long. This is the main problem that the paper solves.

Fig. 1. building dictionary

3. 2 The optimization of LZW

On the strength of that problem, the algorithm should be optimized. In the program with chained lists, because of sequential traversal, it waste some time to find the codes. So we came up with a new method called forward-moving on frequently-used entries. A concrete realization is that adding a new variable
counter to each node for keeping count of the amount of usage. And make the list bidirectional in order to move the node conveniently. When the codes’ amount of usage meeting the specified amount, which is a mark of moving, the node is moved behind the head node (shown in Fig.2.(a)). Thus, nonexpendable codes will be located in the front end of the chained lists, which will accelerate the process to search for codes, thereby, save compression time. Then, we compressed the data of 4.76MB and compared each result. The comparison is shown in Table1.

![Fig. 2. (a)forward-moving on frequently-used entries;(b) forward-moving and sorting on frequently-used entries](image)

The algorithm is optimized further. As the counter comes to the specified amount and the node need to be moved forward, the node is not inserted behind the head node, however, behind the last nonexpendable code (shown in Fig.2.(b)). Thus, the first one coming to the specified amount will be located in the foremost end of the list, the latter ones will be arranged in descending order on the base of amount of usage. This is equal to ordering the nonexpendable codes simply. As a consequence, this method is called forward-moving and sorting on frequently-used entries. This improvement effectively shorten the time. The comparison results is also shown in Table1.

The coding steps of algorithm, which has been optimized, are as follows:

Step1: Initialize dictionary. Dictionary contains all single characters in the data stream.

Step2: Set the specified amount for the mark of moving.

Step3: Set the Prefix P Null.

Step4: Read the next character in the data stream as the current character C.

Step5: Judge whether the string P + C is in the current dictionary.

1) Yes, set P = P + C, that is extending P with C.
2) No.

① Output P’s corresponding code to the encoded data stream. And the counter of P’s corresponding code add 1. Judge the present counter whether meet the mark:

a. Yes, move this node behind the last nonexpendable code.
b. No, do nothing.

② Judge whether the dictionary achieve to the maximum capacity:
If it doesn’t, add the string P + C to the dictionary, otherwise don’t do that.

③ Define P = C. (P only contains C right now.)

Step6: Judge whether there are characters in the data stream:
1) Yes, return step3 to continue the encoding process.
2) No, output P’s corresponding code to the encoded data stream.

End.
Table 1. Comparison of different algorithms

<table>
<thead>
<tr>
<th>Different algorithms</th>
<th>Compression Efficiency (%)</th>
<th>Compression Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic chained lists</td>
<td>77.52</td>
<td>28.425</td>
</tr>
<tr>
<td>Nonexpendable codes move forward</td>
<td>77.52</td>
<td>27.826</td>
</tr>
<tr>
<td>Nonexpendable codes move forward and be ordered</td>
<td>77.52</td>
<td>22.239</td>
</tr>
</tbody>
</table>

Based on the nonexpendable codes move forward and be ordered, we change the specified amount keeping the count of usage, the compression time will also change. The results are shown in Table 2. It is seen from the table that when the mark of moving is 3000, that is to say, simply order the 90% of the codes, the speed of sequential traversal is fastest, and the compression time is the shortest.

Table 2. Comparison of different marks

<table>
<thead>
<tr>
<th>Mark of Moving</th>
<th>Compression Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>21.352</td>
</tr>
<tr>
<td>4000</td>
<td>21.845</td>
</tr>
<tr>
<td>5000</td>
<td>22.239</td>
</tr>
<tr>
<td>10000</td>
<td>23.737</td>
</tr>
<tr>
<td>30000</td>
<td>26.815</td>
</tr>
<tr>
<td>50000</td>
<td>27.513</td>
</tr>
</tbody>
</table>

5 The conclusion

In this paper, LZW algorithm is implemented with chained lists data structure and optimized by the method of putting high-frequency lexical item forward. After comparing and study, it draws a conclusion that using the method mentioned above can shorten the compression time effectively and the time is shortest when the specified amount is an appropriate number. Sorting the high-frequency lexical items that have been put forward is proposed in this paper as a further optimization method. And this method can shorten the compression time further. The greater the amount of compressed data, the effect is more evident. Optimal algorithm can be widely used in bridge vibration data, and other decimal or binary data compression.

References